Kiefer Lam

Registration number 100166387

2020

Real-time Ray Tracing

Supervised by Dr Stephen Laycock

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This document explains how to use the class file `cmpreport.cls` to write your reports. The class file has been designed to simplify your life; many things are done for you. As a consequence some commands presented here are specific to the class file whether they are new commands or customized versions of commonly known LaTeX commands.

Acknowledgements

This section is used to acknowledge whoever's support and contribution. The command that introduces it is ignored in the project proposal, literature review and progress report. It is used in the final report, but is not compulsory. If you do not have an acknowledgements command in your preamble then there won't be any acknowledgement section in the document produced. *Abstract* and *Acknowledgements* sections should fit on the same page.

Contents

1 Introduction **6**
1.1 Aims and Motivations 6
1.2 Problems 6

2 Literature Review **7**
2.1 Project: Real-time ray tracing 7
2.1.1 Brief 7
2.1.2 What is ray tracing? 7
2.1.3 Essential Algorithms 10
2.2 Workflow and plan 11
2.2.1 Display and compute setup 11
2.2.2 Compute kernel 11
2.2.3 Primary rays 12
2.2.4 Tracing primary rays 13

3 Design and Implementation **13**
3.1 OpenCL and OpenGL 13
3.1.1 Host 15
3.1.2 Device 15
3.2 Kernel Structure 16
3.2.1 Ray Trace Kernel 17
3.2.2 Image Kernel 18
3.2.3 Reset Kernel 18
3.2.4 Monolithic Kernel 19
3.3 Ray Generation 19
3.3.1 Primary Rays 21
3.3.2 Reflection 21
3.3.3 Refraction 21
3.3.4 Shadows 21
3.4 Intersection With Objects 21
3.4.1 Sphere Intersection 21
3.4.2 Triangle Intersection 21
3.4.3 OBJ Models 21
3.5 Generating An Image 21
3.5.1 Phong Model 21
3.5.2 Transmission & Reflection 21
3.5.3 Fresnel 21
3.5.4 Shadows 21
3.5.5 Combining Secondary Rays 21

3.6 Acceleration Structures 21
3.6.1 Preliminary Checks 21
3.6.2 Bounding Volumes 21
3.6.3 Grid 21
3.6.4 Bounding Volume Hierarchy 21

4 Results and Testing **21**
4.1 Ray Intersection Tests 21
4.1.1 Ray Tracing with no objects 21
4.1.2 Ray Tracing Spheres 21
4.1.3 Ray Tracing Triangles 21
4.1.4 Acceleration Structures 21
4.2 Resolution 21

5 Future Work **21**

6 Conclusion **21**

References **28**

List of Figures

1 Diagram of the implementation architecture design 15
2 Sphere with Phong shading 20
3 Sphere with reflective material 21
4 Sphere with transparent material with refractive index 1.517 22
5 Sphere with transparent material with refractive index 1.04 23
6 Spheres with increasing refractive index ranging from 1.0 -> 1.7 23
7 Transparent sphere with refractive index 1.517 with fresnel effect 24
8 Transparent sphere with refractive index 1.04 with fresnel effect 25
9 Sphere casting a hard shadow on a surface 26
10 Shadow with anti-aliasing 27
11 Sphere casting a soft shadow on a surface 27

1 Introduction

1.1 Aims and Motivations

Ray tracing has already been established as a viable technique to produce realistic graphics in computer generated graphics by simulating light rays. The aim for the project is to be able to produce visually realistic images quickly, sequentially, and continuously to create a real-time experience. A real-time application is subject to real-time constraints where the application must produce a response to an event within a time constraint. In this case, the application must produce an image in time after an event such as user input. There are many use cases for this project including:

- Realistic visuals in 3D games
- Faster rendering for computer generated imagery in video
- Simulating light rays and how they interact with objects in a scene

1.2 Problems

Many industries already use the ray tracing technique to produce realistic visuals in games or video however the universal issue that comes with ray tracing is performance. Since the technique requires a significant number of rays to produce a realistic and accurate (not noisy) image, it often takes a long time for the rendering process. There is usually a trade-off between visual fidelity and performance (Vasiou et al., 2018).

Video Real time ray tracing is less of a problem for video as the experience is not affected by the speed at which the video was rendered. Because of this, ray tracing in CGI uses more rays to create a more realistic image. However it would still benefit from performance during production.

Games Video games are an example of something that would benefit more from real-time ray tracing rendering as the experience is generally better and smoother when the image is rendered quicker. Games prioritise performance over realism as the game would be 'unplayable' if it is unable to produce the image with a quick response to user input. This is why games will usually use less rays at the expense of realism.

The number of rays is not the only factor affecting performance. Each individual ray has to perform an intersection test with every object in the scene. Simple objects such as spheres and planes will only require a single intersection test however complex models will require multiple intersection tests for each part of the model.

2 Literature Review

2.1 Project: Real-time ray tracing

2.1.1 Brief

In this report, I will discuss the ray tracing technique, its use in the project, the differences with rasterization, and the approaches to be used in the project.

2.1.2 What is ray tracing?

Computer Generated Image Ray tracing is a technique used to computer graphically generate a 2D image of a 3D world typically viewed on a flat screen. The technique is commonly used to produce a high degree of visual realism in computer generated images.

Conventional Rendering Technique Since ray tracing is relatively computationally intensive, rasterization is the conventional rendering technique used in mediums that require images to be rendered quickly with low response times to user input (low latency) such as video games. Rasterization generates the 2D image of the 3D world by projecting the fragments computed from a mesh with vertices and faces onto a plane which is then displayed onto the screen. However, ray tracing simulates the real world physical process of how objects are illuminated by light rays in a virtual world. To produce an image with high visual fidelity, an innumerable amount of rays must be used. Depending on the complexity of the scene, millions of rays may have to be generated to produce a single image.

Realtime-ness For the project to run in real-time, the system must guarantee response to events with low latency. The latency of the response must be in the order of milliseconds for the system to be real-time; more so in applications such as video games where a high response time will result in a complete failure as a game. An example in a video game would be: the user inputting an event to move a character and the response being the system displaying an image with the character in a different location appearing to move. The realtime-ness of the application will largely depend on the time it takes for the image to be rendered as it is the main process causing the latency.

Ray tracing approach To produce the image, a ray is created for each pixel of the image. These initial rays are referred to as primary rays and are traced outwards from the camera origin. This approach is modelled after the pinhole camera where light rays pass through the pinhole and land on the film (screen) at the location corresponding to the angle of the ray (Glassner, 1989, 1.1.2). It would be unreasonable to simulate

this by tracing all light rays originating from every light source and capturing the light rays that happen to land on the screen. This is forward ray tracing. Instead, we would use backward ray tracing the rays are traced from the screen and towards a light source (Glassner, 1989, 1.2). The light sources may sometimes be other objects and not necessarily a true light source and the bounces of rays are usually limited to a relatively small number (relative to the true actual number of bounces of light rays in the real world). The purpose of these limits are to impose a time constraint so that the result is produced in a reasonable time frame making it smoother. This will reduce the potential degree of realism that can be achieved however it is necessary to ensure the system runs in real-time. The most basic ray tracer would return the colour of the closest object hit by the primary ray which would apply as the colour of the pixel for the ray. However this does not produce a very realistic image; more rays must be traced, originating from the intersection point, to create a more realistic image with lighting and shadows.

Secondary rays Rays stemming from the intersection point of the primary ray are referred to as secondary rays. Lighting, shadows, reflection, and refraction are some of the features that can be simulated with secondary rays. When the primary ray intersects with an object, more rays can be generated with the intersection point as the origin. The direction of the ray may be different depending on the information we are trying to find. Secondary rays can be generated and processed recursively to simulate light bounces.

Shadow rays To determine whether the primary ray intersection point is in a shadow, a secondary ray can be generated and traced towards light sources where the point would be in a shadow if it intersects with an object before the light source. The result from the primary ray can then be modified to make the pixel appear darker and in effect, in the shadow.

Reflections When reflection rays are generated, the direction of the new ray should be reflected about the normal of the surface of the intersection point. The outward angle from the normal should be the same as the angle of incidence of the previous ray. The new ray can be treated as a primary ray and have its result mixed with the parent ray with varying degrees depending on the reflectivity of the objects material. For example, a mirror would have more weight on the colour value of the reflection ray than the primary/parent ray.

Refraction The basic idea of tracing refraction is also based on refraction in the real world where physical concepts are considered instead of brute force tracing the ray, stepping through the object until the ray exits. If the object material information has a refractive index, we can use this information and the angle of incidence to find the angle

at which the light ray 'travels' through the object. This method assumes the density of the object is constant.

Combining rays (Glassner, 1989, 1.2.3) To determine the colour of the pixel, we have to combine all the rays that the pixels corresponding primary ray and all of its secondary rays to produce a sensible colour to be displayed on the screen. For example, a red and green ray should combine and produce the colour yellow. We also have to take into account that light rays usually lose energy when they bounce around in the real world. This can be achieved by reducing the weight of a rays impact on the final result depending on the number of bounces before the secondary ray. Shadows, reflections, and refractions will require different algorithms for combining their results into the final result. Physically, these are all the same type of ray, but it is convenient to classify them as different rays computationally. The combining algorithm will be more complex than it may seem at first. When real world physical light rays combine, their waves and amplitude superpose in an additive manner. If we apply that simple additive method to the ray tracer, we would end up with white most of the time as we add the ray colours and get values above 1.0 in the RGB channels. We could increase the dynamic range of the rays by using the highest value in the scene and use that as the ceiling, scaling every other value down proportionally. However this requires every pixel to be traced beforehand.

Aliasing (Glassner, 1989, 1.4) The signal our eyes receive from light rays are continuous. There are no pixels or grids in the human eye so everything we see is smooth with great detail. The issue with modern computers is that they cannot represent a continuous signal. Digital images are made up of a grid of pixels where the size of a single pixel is the limit of how fine of a detail we can display. Because of this, the image produced will have an aliasing effect where a smooth edge appears jagged. To counter this, anti-aliasing techniques are used. One approach of anti-aliasing is tracing multiple primary rays per pixel where pixels are split into sub-pixels. This is referred to as multi-sampling (multiple samples for each pixel) and the final result of the pixel is a combination of the sub-pixels. The easiest way to achieve this would be to just produce an image with a higher resolution and scale it down when finally displaying the image, letting the display software decide on how to interpolate the neighbouring pixels. While it is the easiest and simplest approach, it is also the least efficient; there are more optimised techniques to achieve anti-aliasing faster and sometimes better. Instead of blindly tracing extra rays from each pixel, we can select the only the pixels which reside on the edges of objects to trace the sub-pixel rays as surfaces generally do not need anti-aliasing.

2.1.3 Essential Algorithms

Ray intersection algorithms In order to detect whether a ray hits an object in the scene, we'll need to test whether the ray, as a straight line, intersects with the object. With only the ray-triangle intersection test, we can test every and any arbitrary shape as the shapes can just be converted into a triangle mesh. However this is inefficient as models and meshes could have a large number of triangles so it is a good idea to include tests for commonly used shapes such as spheres and planes.

$$R = R_o + t(R_d)$$

The ray R is denoted by the ray origin R_o plus the ray direction R_d where R_o and R_d consist of three components (x, y, z). Any point on the ray can be calculated by adjusting t where $t > 0$. For values of $t < 0$, the point is behind the ray origin and therefore disregarded.

Spheres (Bourke, 1992) Spheres are commonly used in ray tracing to test the ray tracing techniques as the every point on the surface of a sphere has a different normal. The intersection test algorithm is also relatively quick and simple compared to a triangle mesh.

A sphere S with a center (S_x, S_y, S_z) and radius S_r can be described by

$$(x - S_x)^2 + (y - S_y)^2 + (z - S_z)^2 = S_r^2$$

To solve the intersection problem, we can substitute the ray/line equation into the sphere equation as such:

$$(R_{ox} + t(R_{dx}) - S_x)^2 + \tag{1}$$
$$(R_{oy} + t(R_{dy}) - S_y)^2 + \tag{2}$$
$$(R_{oz} + t(R_{dz}) - S_z)^2 = S_r^2 \tag{3}$$

Then we can rearrange this in terms of t and simplify to:

$$at^2 + bt + c = 0$$

where

$$a = R_{dx}^2 + R_{dy}^2 + R_{dz}^2 = 1 \tag{4}$$
$$b = 2(R_{dx} \cdot (R_{ox} - S_x) + R_{dy} \cdot (R_{oy} - S_y) + R_{dz} \cdot (R_{oz} - S_z)) \tag{5}$$
$$c = (R_{ox} - S_x)^2 + (R_{oy} - S_y)^2 + (R_{oz} - S_z)^2 - S_r^2 \tag{6}$$

The coefficient a equation can be recognised as the square of the length of the vector R_d and since the ray direction is normalised, the coefficient is always equal to 1.

The solution for the quadratic is:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If the discriminant $b^2 - 4ac$ is negative, there are no solutions and hence the ray does not intersect with the sphere. If the discriminant is 0, there is only 1 solution where the ray is tangent to the sphere. Else there are two solutions of t where the ray enters the sphere and another where the ray exits the sphere. If any of the solutions of t are negative, we can ignore those values as they are behind the ray origin. To find the intersection point after evaluating t, we just need to input the value of t into the ray equation $R = R_o + t(R_d)$. For most cases, only the smaller value of t is relevant as it is the closer intersection point.

2.2 Workflow and plan

For the early stages of development, there is a general order of the steps and features to develop in order to create a basic functioning ray tracer and display the results on the screen.

2.2.1 Display and compute setup

The first step in the project is to be able to display an image on the screen. OpenGL will be used to store and display images as a texture. For the actual computation, OpenCL will be used to accelerate computation via parallel processing. Since both platforms have their own contexts, they also have their own memory buffers so to display the results from the OpenCL kernel, we'd have to move the data from the OpenCL buffer to the host (CPU/Primary) memory and then upload the data to an OpenGL buffer/texture. This has a huge impact on performance as the data transfer between host and client comes with an overhead. With OpenCL and OpenGL interoperability, the two platforms can share memory on the GPU so that the OpenCL kernel can write directly to an OpenGL texture. To set this up, the OpenGL context must be provided to the OpenCL platform via a properties array during the creation of the OpenCL context.

2.2.2 Compute kernel

(Munshi et al., 2011) OpenCL kernels are functions written in the OpenCL C programming language that acts as an entry point whenever it is queued into the command queue. Source code is passed through to the OpenCL platform where a 'program' is created and compiles the source code. Kernels can be declared on the host afterwards

using the name specified in the code where they are queued from the host side whenever they are supposed to be executed. Usually, kernels will have an input and output specified in the kernel function parameters both of which are either primitive values or memory buffers on the GPU. In most cases, the input will be a memory buffer with data copied from the host memory and the output memory buffer will be written to by the kernel and then copied to the host memory as the result. For ray tracing, the input to the kernel will be a memory buffer with the world information such as the objects, lights, and their materials while the output will be an image type where the kernel will write specific colour information directly to the image. With OpenCL-OpenGL interoperability, the image memory does not have to be transferred to the host memory and can simply be reused by OpenGL as a texture to be displayed on the screen.

With the OpenCL platform setup, the next stage would be to write the kernel that deals with the ray tracing. On the host side, there would be a struct with the necessary world information that is identical to the struct defined in the OpenCL source code. The struct must be aligned with 8 bytes to ensure it is copied across correctly. On the host side, the kernel can be queued to execute n amount of times and in two dimensions. For example, if our image resolution was 1280 by 720, the two dimensional range would be [1280, 720]. From within the kernel, the program can access the ID of the kernel which is unique for every instance within the range. With a two dimensional range, the kernel can access two ID's which can be represented as the x and y coordinates of the image which can then be used to calculate the ray origin and direction.

2.2.3 Primary rays

With the kernel setup, we can begin creating the primary rays. The primary rays can be created with the origin at a virtual screen at a distance d away from the camera origin. This creates a clipping effect like a the near plane of the frustum in a perspective projection. The direction of the ray can be calculated by finding the vector from the camera origin to the pixel of the virtual screen in world space. With the current coordinates for the pixel, the top left corner of the image $(0,0)$ would become the ray pointing directly forward. In order to make the center pixel of the image point directly forwards, the pixel coordinates will have to be normalised and centered. To do this, we can simply divide the pixel coordinates by the resolution of the image and subtract 0.5. This would give us a range from $-0.5 \rightarrow 0.5$ in both x and y coordinates. To make it easier to work with, we can then multiply the values by 2 to give it a range from $-1.0 \rightarrow 1.0$. However this does not take into account aspect ratio. With these coordinates, the end result would appear stretched if the image resolution was wide. To accommodate for this, the x coordinate is multiplied by the image width divided by the image height. For example, with a resolution of $[1280, 720]$, the x coordinate will range from $-1.778 \rightarrow 1.778$ (to 4 s.f).

With the image dimensions (width, height) represented by I_w and I_h, and the pixel coordinates represented by I_x and I_y the primary ray can be calculated as such:

$$A = \frac{I_w}{I_h} \quad \text{Find the aspect ratio} \quad (7)$$

$$N_x = 2(\frac{I_x}{I_w} - 0.5) \quad \text{Normalised X origin value} \quad (8)$$

$$N_y = 2(\frac{I_y}{I_h} - 0.5) \quad \text{Normalised Y origin value} \quad (9)$$

$$R_o = \begin{bmatrix} N_x \times A \\ N_y \\ d \end{bmatrix} \quad (10)$$

$$R_d = \frac{R_o - C_o}{|R_o - C_o|} \quad C_o \text{ represents the camera origin} \quad (11)$$

2.2.4 Tracing primary rays

After calculating the primary rays, the next step would be to trace them. For a basic ray tracer, the simplest way to do this would be to just loop over every sphere in the scene and perform the sphere-ray intersection test and return the colour information of the closest sphere hit. If the ray does not intersect with anything, return the background colour.

Once the basic ray tracer has been implemented, the next steps can be any of the features mentioned above in almost any order because they would mostly only depend on the primary rays as they generate the secondary rays themselves.

3 Design and Implementation

3.1 OpenCL and OpenGL

At the basic level, the implementation is split into two stages: *initialisation* and *main loop*. During initialisation, the OpenGL and OpenCL environments are setup. The only task OpenGL is responsible for is displaying the image after it's produced by the ray trace so the initialisation consists only of creating the window (via GLFW), creating an image buffer on the GPU, and displaying the image once it has been written to.

For OpenGL-OpenCL interop, the OpenGL context is created first and then passed to the OpenCL environment as an argument to the context properties when the OpenCL context is created. This allows OpenCL to directly access memory buffers created by OpenGL on the GPU. Since the OpenGL side of the application is only interested in displaying the image, the only memory buffer OpenCL accesses it the image buffer. Both OpenGL and OpenCL creates *handles* to memory buffers on the GPU as the memory

on the GPU (device) cannot be directly accessed by the CPU (host). In this stage, there are separate handles to the same device memory buffer for OpenGL and OpenCL.

The application will prompt the user to select which device and platform to use if there are options available. When there are no options, the application will automatically select the only valid configuration of device and platform. The version of OpenCL will depend on the device. The version of OpenCL used throughout the project is OpenCL 2.1.

OpenCL *Programs*, *Kernels*, and memory buffers are also setup during this stage. The OpenCL kernel code, coded in the OpenCL C programming language which is based on the C99 specification with some C11 features (Howes and Munshi, 2015), is loaded from the independent source files and compiled during runtime. The kernel programming language allows the use of the C 'include' directive so only a single source file code is passed to the OpenCL compiler where the compiler will load the specified kernel files. This is found in the 'kernel.cl' file. The source code is compiled into an OpenCL program which contains the kernels declared in the source code. Kernels are functions declared in a program where memory buffers can be passed to as arguments and act as the entry point for execution when the host queues the kernel to the device. (Howes and Munshi, 2015)

With the interop environment initialised, the OpenCL command queue is created. The command queue is the main event queue where events are queued by the host and consumed by the device. Interactions between the host and device mainly use the command queue during the main loop. Kernel queuing and memory buffer data read/write operations, are handled by the command queue. Memory buffers should only be created once as allocating memory is relatively slow so it should be avoided during the main loop. For this reason, the application only creates/allocates memory buffers during the initialisation stage and the only memory events that occur inside the main loop are memory updates (e.g. for updating object positions or camera positions) and kernel queuing.

It is possible to specify a blocking flag for some events when they are queued (Howes and Munshi, 2015). The OpenCL queue and event architecture allows the host side to manage the concurrency of events that are queued. It is unimportant that events that are queued and do not depend on each other to run or finish in the order that they were queued however there are some events that do rely on a related event to finish first. I.e. in this application, the ray trace kernel event must be completed before the image kernel event is queued. By default, kernels queued are not *blocking* so the OpenCL context is free to run the next queued event however there are also functions for the host to wait for a previously queued event to finish before queuing another. Queuing commands is generally quick but it is the execution of the events that will take time, hence the host can queue a kernel and wait for the kernel to finish before queuing another. In the implementation, the ray trace and image write kernels are queued dependent on each other however the event updating scene information is non-blocking because the scene

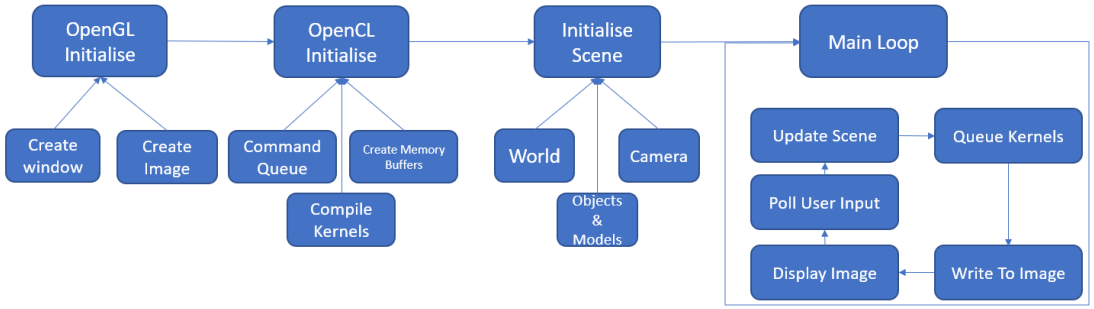

Figure 1: Diagram of the implementation architecture design

is not always changing so this task is sometimes skipped entirely.

Figure 1 shows a diagram of the implementation architecture design. The main loop is mainly short and simple as most of the operations handling ray tracing and generating an image reside in the kernel code.

3.1.1 Host

Device memory usage in kernels must be managed by the host because there is no way to dynamically allocate or de-allocate memory in kernels. An alternative to this would be to create new memory buffers on the host side and set a kernel argument to the newly created memory buffer however this is slow and would likely have a significant impact on the time spent in the main loop. Due to the nature of this, the memory usage is generally static and any apparent additions in memory, e.g. adding an object to the scene, is usually a memory update in a pre-allocated memory buffer. For example, the host may pre-allocate a memory buffer large enough to store 1000 spheres so the device program will always be using at least 1000 spheres worth of memory even if the entire memory block are $0's$ or the kernel never uses the memory buffer.

The device also has no control over data transfer between device memory and host memory. The device can read and write memory between device memory in the form of memory buffers however only the host can manipulate data between host memory and device memory. Because of this, there is less flexibility with code in OpenCL.

3.1.2 Device

When memory buffers are created via the host, the host must specify an address space in which the memory buffer is created in and can be one of the following: global, local, constant, private. Depending on the OpenCL implementation, the different address spaces could offer major performance benefits and could be physically allocated in dif-

ferent physical memory or memory types. Memory allocated in the constant address space are read-only to kernels and may be cached more aggressively, providing a performance boost to access to the cached memory. Global memory is generally the slowest address space as OpenCL must allow the kernels write access to the memory. Constant and Global memory can be accessed by all work-items of a kernel during its execution however since memory in the constant address space will be unchanging between work-items, its contents can be cached. Local an private memory are generally declared inside the kernel function itself with no interaction with the host. Local memory can be accessed by all work-items in a work-group however private memory can only be accessed by the work-item. To increase access performance, if a kernel requires multiple reads from global memory, it is copied to private memory via code where subsequent accesses to the required data is done via the private copy. Obviously, this solution will provide any performance benefit when writing to global memory. When a memory buffer is shared with the OpenGL context, whenever OpenGL accesses the memory buffer, the data in the buffer will always be up to date as the actual memory on the device that both the OpenCL and OpenGL handles point to are the same. This is evident as the OpenGL image is displayed immediately after the OpenCL kernel responsible for writing to the image memory buffer is executed.

The OpenCL C programming language, while based on C99, has some limitations due to the fact that memory cannot be allocated during kernel execution and functions undergo a similar optimisation to loop-unrolling. As a result, recursive function calls are forbidden and control flow is limited. This poses a problem for recursive ray tracing where the results of the rays traced must be concluded backwards (from the tail end). The workaround used was to use a statically allocated array acting as a heap to process rays in a recursive fashion iteratively. (Gaster et al., 2012) While there are some limitations and little flexibility, OpenCL C does include some quality of life advantages with built-in functions and data-types that align with the type of processing required by ray-tracing. The built-in functions include *dot*, *cross*, and mathematical functions such as *log* and *sqrt*. Alongside regular scalar data types, OpenCL C also includes vector data types which can conveniently represent data constructs such as positions or directions. I.e. a ray is a structure composed of two *float3* representing the ray's origin and direction.

3.2 Kernel Structure

Designing a kernel architecture for ray-tracing which takes advantage of multiple kernels is a non-trivial task. While a single monolithic kernel can be used to execute the whole process, from ray-tracing to image and colour processing, it is generally a better idea to separate parts of the process into individual kernels where independent code and relevant memory are isolated (Laine et al., 2013). When code performing independent tasks are separated into individual kernels, the code base is generally more organised

easier to refactor if needed. Another benefit is that having multiple kernels naturally encourages vectorisation in the processing of data instead of using branching inside a single kernel. The ray-tracer implementation uses two kernels, a kernel for tracing rays, and a separate kernel for processing the results of the trace and writing the produced colours to the image.

The command queue allows kernels to be queued with *ND* range where *ND* represents the dimensions of the range (e.g. 3D for a three dimensional range) and in an *ND* range kernel execution, each work-item is given *N* ID's which correspond to the range. E.g. a $2D$ range of 3×3 will result in 9 work-items executing the kernel with two ID's $(0, 1, 2)$ in each dimension. During kernel execution, the work-item can query its global ID which can be used to access data in locations corresponding to its ID's.

3.2.1 Ray Trace Kernel

The ray-trace kernel is responsible for generating rays and tracing them by running intersection tests with objects in the scene. No colour processing or image generation is executed in this kernel; the kernel only handles tracing rays and storing the results in a memory buffer that will be accessed by a separate kernel to generate the image. The kernel is queued with a $2D$ range of the image resolution width by the image resolution height which will allow for easy method for the work-item to identify which pixel it should trace rays for as the kernel can simply query its global ID's to find which pixel it is for without any additional processing.

Since the image generation is not happening in this kernel, the results produced when the rays are traced must be stored in a memory buffer for a different kernel to use. To store results of a ray that is traced, a C struct was created to encapsulate the data produced such as the intersection point, surface normal, object index, object type. The global memory buffer that acts as storage for the trace result data is an array of the result struct with varying array length depending on the number of bounces and the number of secondary rays that are passed to the other kernel. The array length can be calculated by $W \times H \times N$ where W and H are the width and height of the image and N is the maximum number of rays traced including the primary ray and any subsequent secondary rays. N can be calculated with

$$\frac{1 - C^{B+1}}{1 - C}$$

where C is the number of child rays (child rays are rays stemming from a primary ray or secondary ray whereas secondary rays are all other rays that are not the primary ray) and B is the number of bounces. To get the starting index in the array for each pixel, the same formula can be used, replacing W and H with X and Y which are the global ID's in each dimension. The formula is based on the summation formula for geometric progression where the first term is 1 (Rosen, 1999).

3.2.2 Image Kernel

When all the rays have been traced, the information retrieved from the rays can be used to create an image based on the path the ray travelled and the objects it intersects with each bounce. The global memory buffer used in the *ray-trace kernel* is passed to this kernel as an argument as *global* as it is in the global address space however the buffer only needs to be read from in the image kernel. The kernel requires multiple accesses to the global memory buffer and global memory access is slow so the each of the kernel work-items copies a slice of the global memory buffer that it will read from. The image kernel is queued with the same *ND range* as the ray-trace kernel so the global ID's will be the same for the same working pixel and so the formula used to calculate the index offset in the global memory buffer for each work-item is the same; this is used to find the starting location of the slice when copied from the global memory buffer into the work-items own private memory and using N, as previously mentioned, for the length of the slice.

A tree is used to structure the rays and their children in the global memory buffer array. The colour result of any ray is dependent on the colour result of all child rays so the kernel must calculate the colour results starting with the deepest node in the tree. Thus, the kernel uses a depth-first approach when traversing the tree. A node's final colour result is a mix of its own material and the colour results of its child rays. If the child index for each type of ray (reflection, refraction, etc) is consistent for each node, then the index of the node can be used to identify its type which affects how the the colour is mixed into the final colour result of the node. If the ray doesn't intersect with an object, it queries a cubemap texture for its result colour based on the ray's direction which represents the sky/environment.

Overall, the only tasks performed in this kernel is to determine the colour of the output based on the materials of the surfaces that the rays intersect. No actual ray-tracing is performed during the execution of this kernel.

3.2.3 Reset Kernel

A third kernel is used during the process to ensure that in the next iteration of the main loop, the working global memory buffer that stores the ray-trace results structs is cleared. The kernel is queued with a 1D range from 0 to L where L is the length of the global memory buffer. While the result struct is relatively large, not a lot of data must be written as only a single boolean flag must be written to for each member of the array. The purpose of the boolean flag is simply to indicate whether the the ray has been traced as some rays will be omitted if they do not need to be traced. E.g. when the intersecting object material is completely opaque, a refraction ray does not need to be traced.

3.2.4 Monolithic Kernel

If a single kernel is used for the entire process, the need to store data into a global memory buffer is removed as the work-item will have all the data necessary and the data will be encapsulated within the work-item. The only memory buffers that need to be created on the host side will be the scene objects and the final image output. The storage for the ray-trace data will be local variables inside the kernel because there is no need for access to the ray-trace data outside of the kernel.

Depending on the hardware, there is also a limitation on the size of memory buffers less than the total memory of the device. If the bounce limit was too high or the number of child rays was too high, the size of the global memory buffer may be higher than the limit and would prevent the buffer from being created. However there is not a limit to the number of local variables in a work-item (or the limit is sufficiently large) and the size of the array would be substantially less as it only has to contain the data required for a single work-item, much less than a single buffer for the entire image.

This kernel structure is not encouraged (Laine et al., 2013) as it naturally results in less organised code, maintainability, and may result in a lower performance. There is also less flexibility on profiling and event control as the entire ray-tracing to image process becomes one single event on the command queue.

3.3 Ray Generation

In the ray-trace kernel, rays are generated by computing their origins and directions with each intersection or by the view in the case of primary rays. Unlike the real world, the rays will originate from the view/eye and be traced outwards; this is backward ray tracing. The method of computing the starting position and direction of a ray differ with the intersect surface material. Typically, a child ray's origin is the at the intersection point of the parent ray and the direction is some function of the parent ray direction.

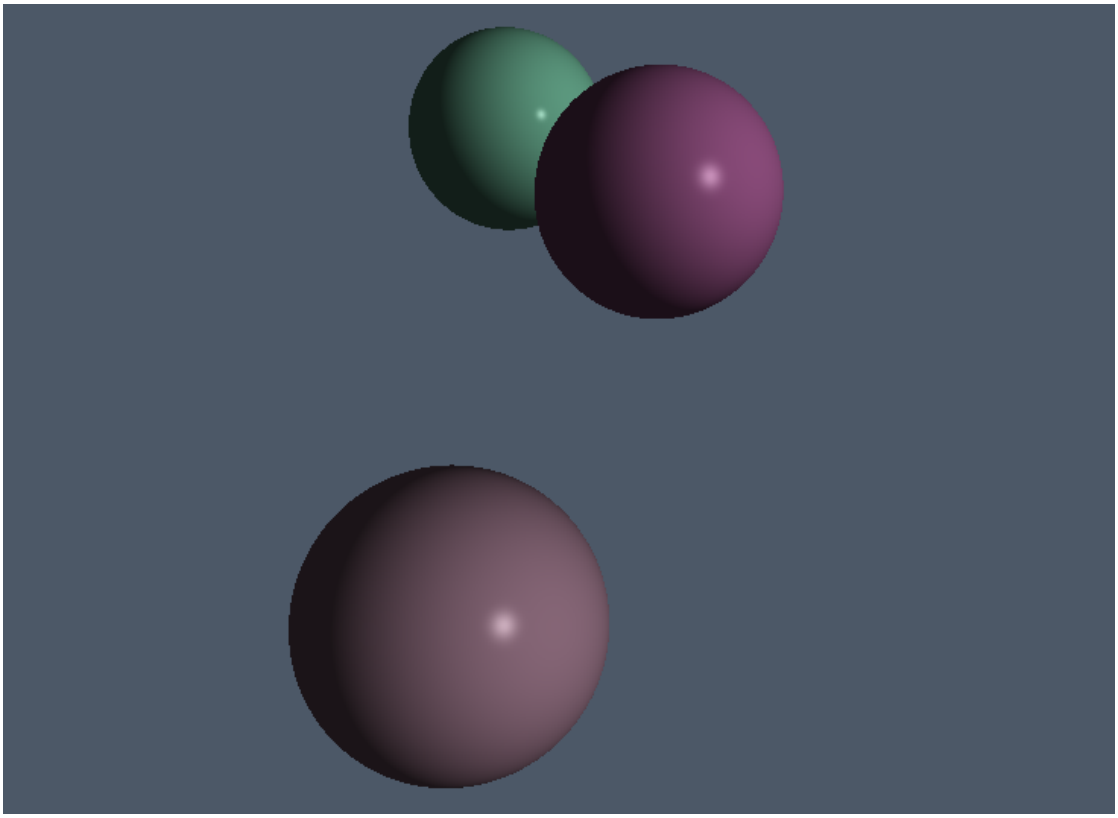

Figure 2: Sphere with Phong shading

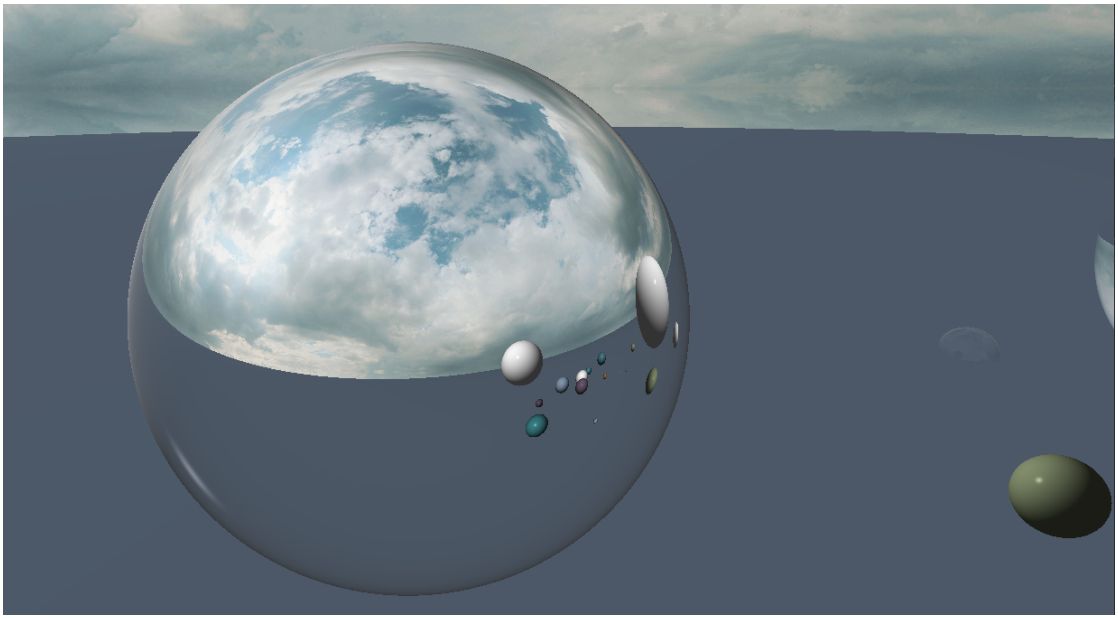

Figure 3: Sphere with reflective material

3.3.1 Primary Rays

3.3.2 Reflection

3.3.3 Refraction

3.3.4 Shadows

3.4 Intersection With Objects

3.4.1 Sphere Intersection

3.4.2 Triangle Intersection

3.4.3 OBJ Models

3.5 Generating An Image

3.5.1 Phong Model

3.5.2 Transmission & Reflection

3.5.3 Fresnel

3.5.4 Shadows

3.5.5 Combining Secondary Rays

3.6 Acceleration Structures

3.6.1 Preliminary Checks

3.6.2 Bounding Volumes

3.6.3 Grid

3.6.4 Bounding Volume Hierarchy

4 Results and Testing

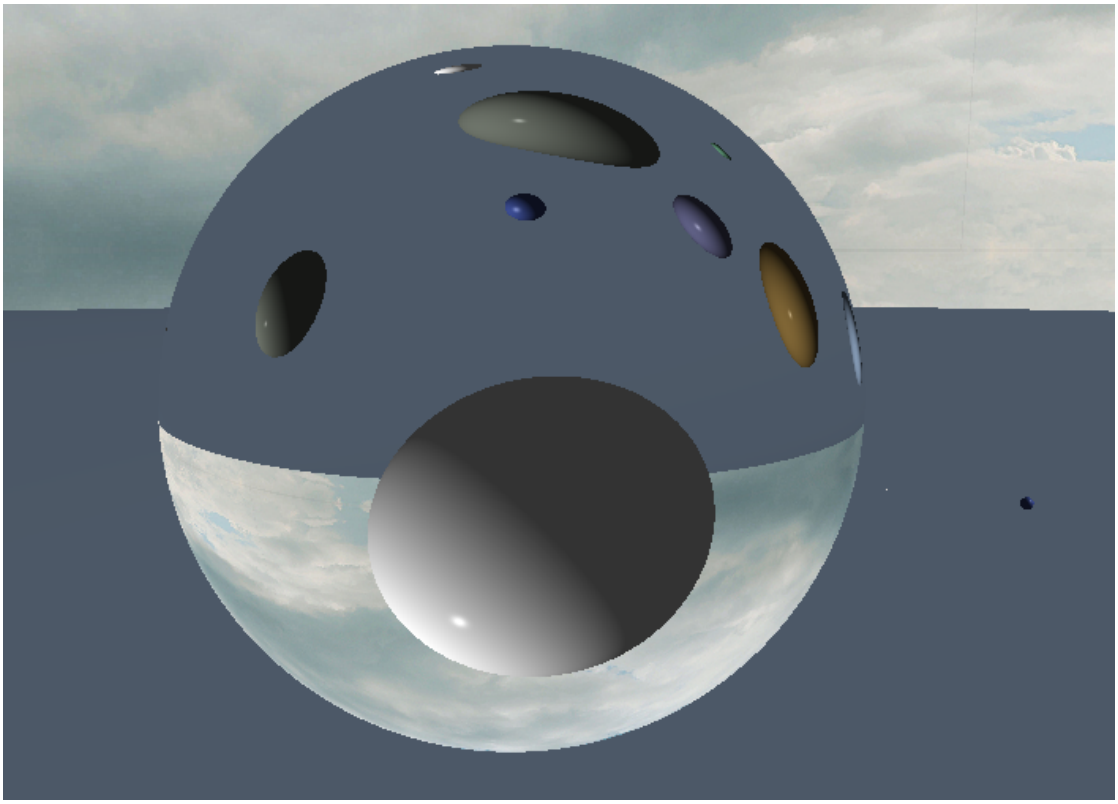

Figure 4: Sphere with transparent material with refractive index 1.517

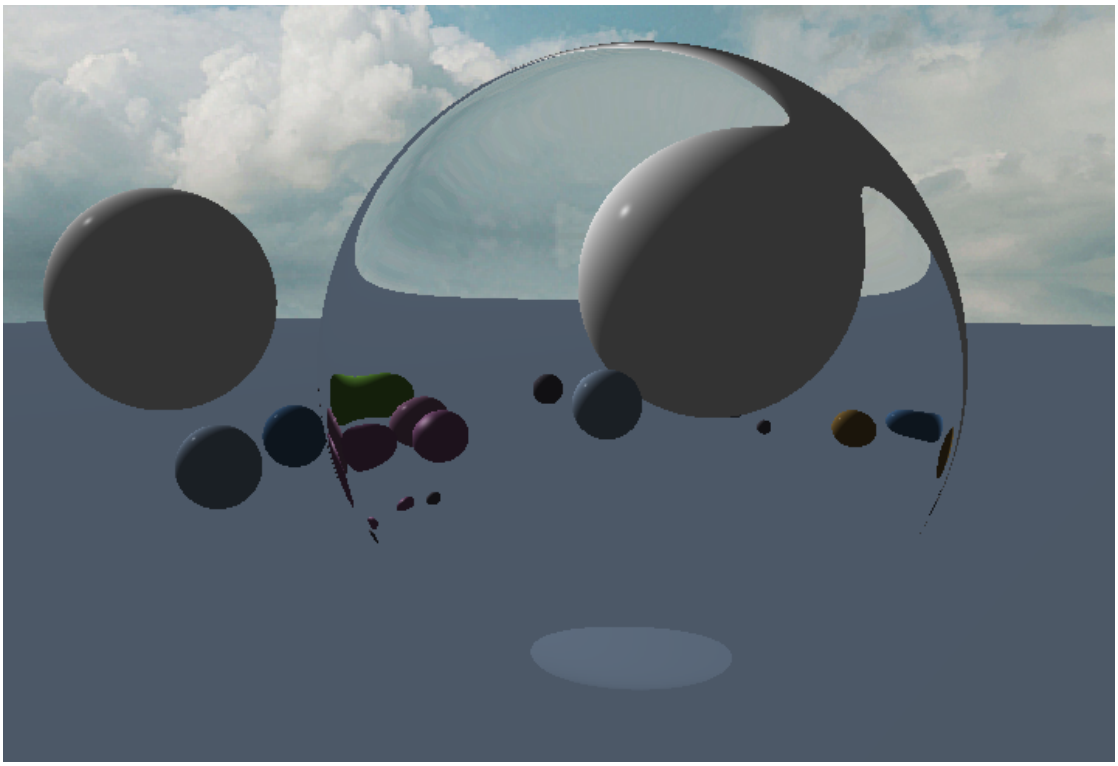

Figure 5: Sphere with transparent material with refractive index 1.04

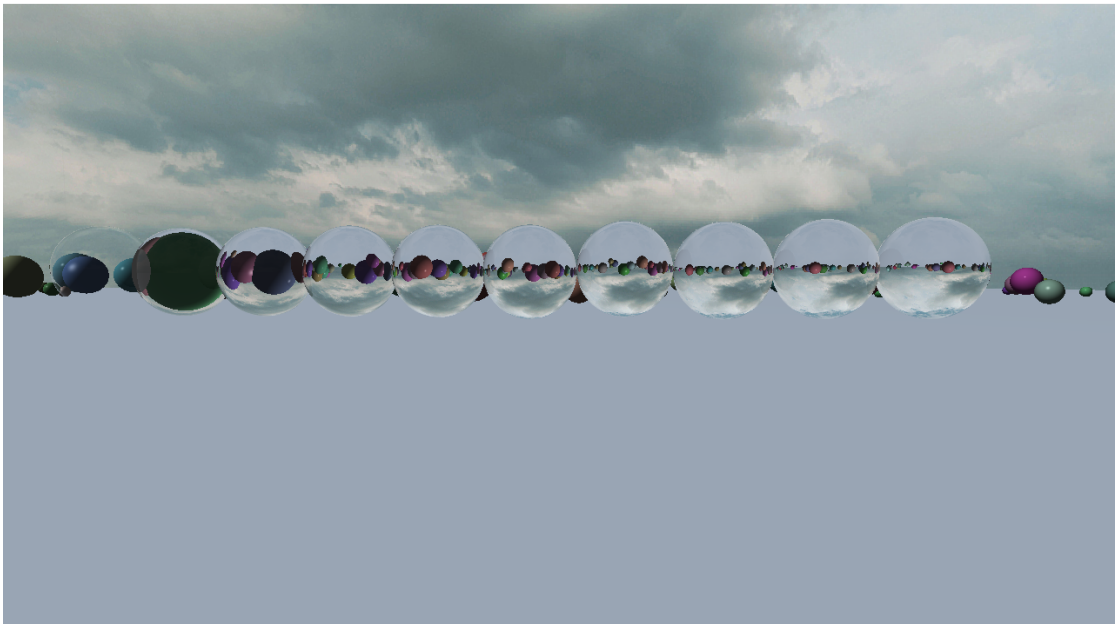

Figure 6: Spheres with increasing refractive index ranging from 1.0 -> 1.7

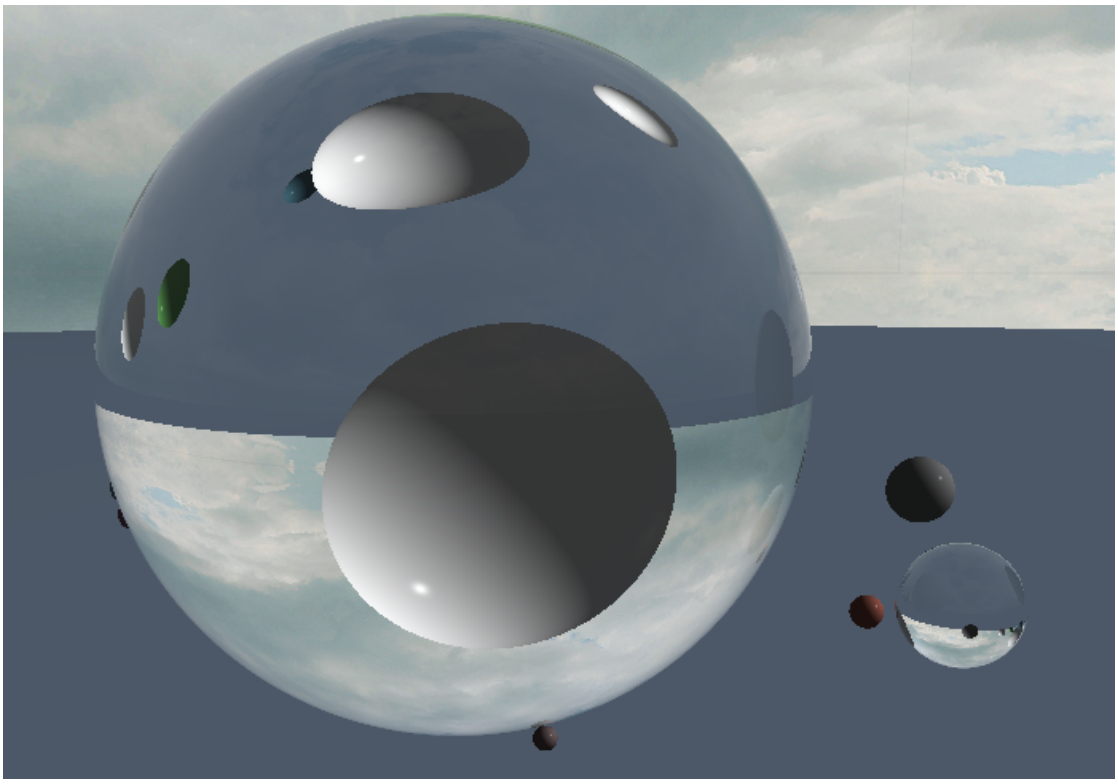

Figure 7: Transparent sphere with refractive index 1.517 with fresnel effect

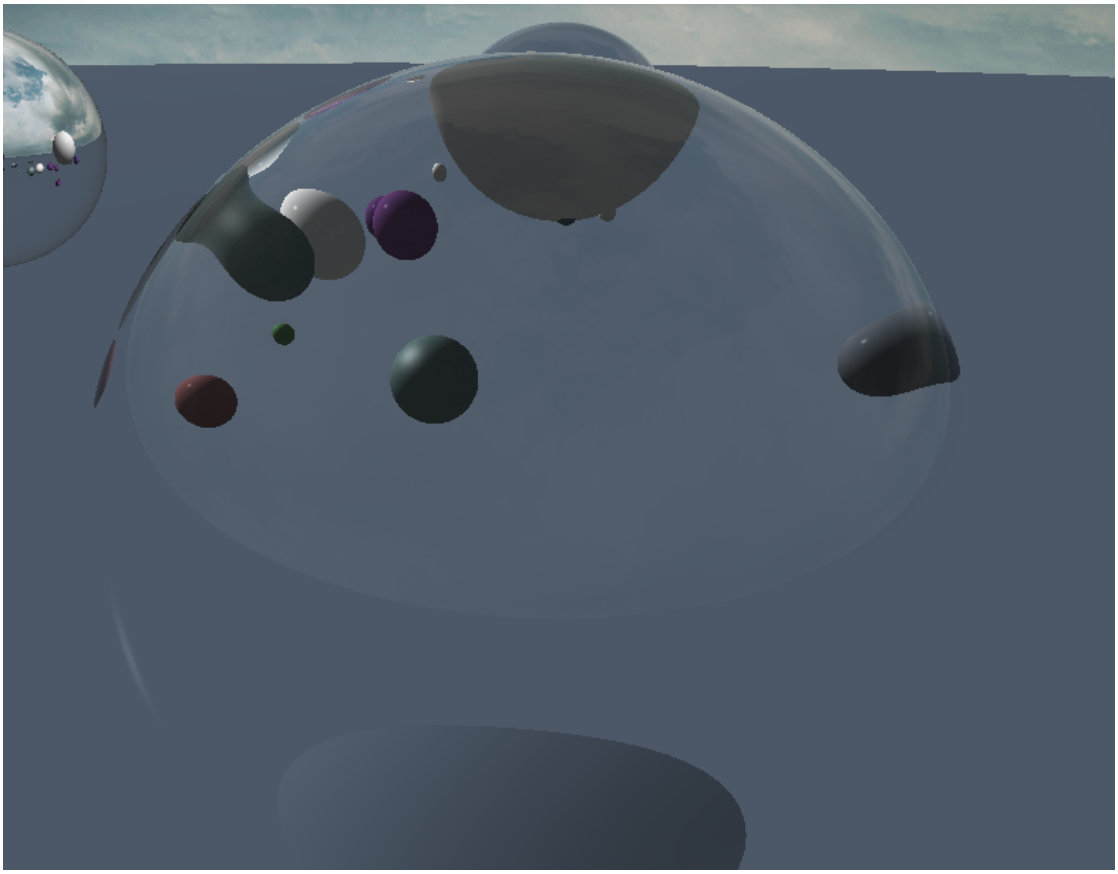

Figure 8: Transparent sphere with refractive index 1.04 with fresnel effect

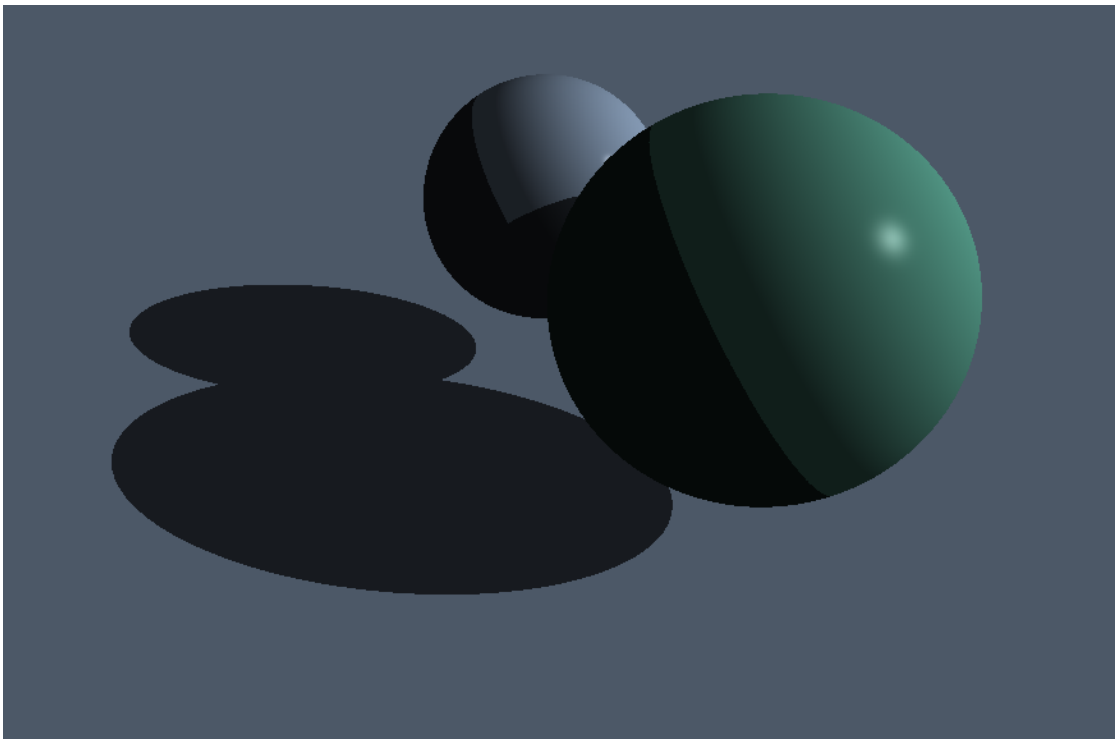

Figure 9: Sphere casting a hard shadow on a surface

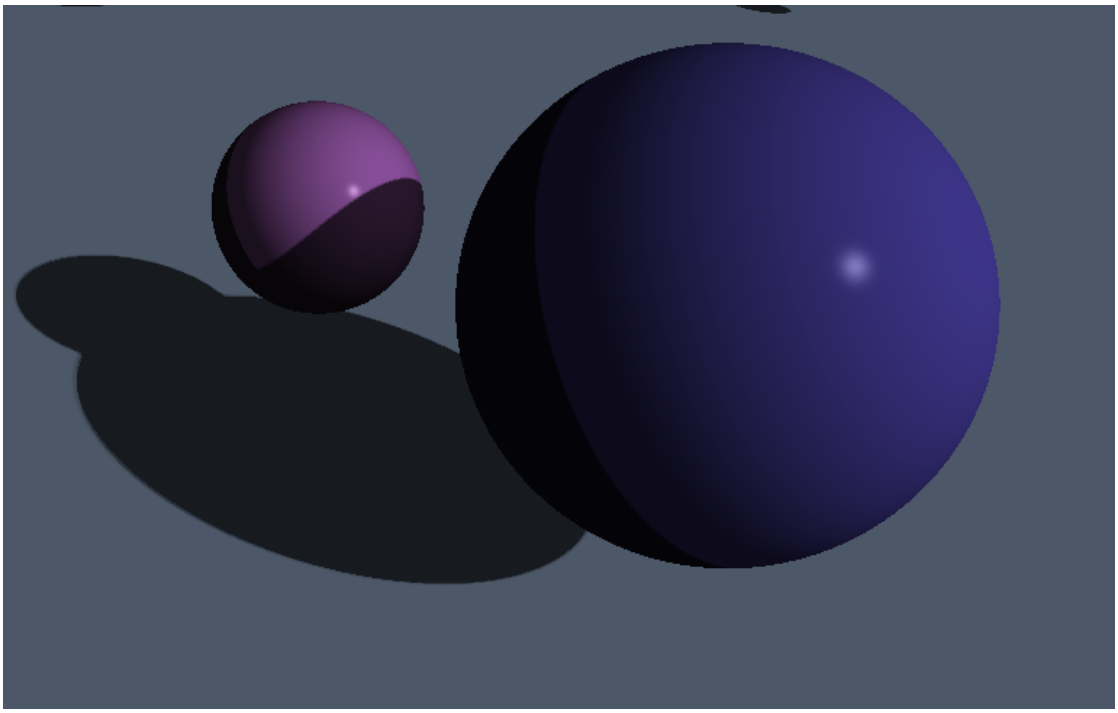

Figure 10: Shadow with anti-aliasing

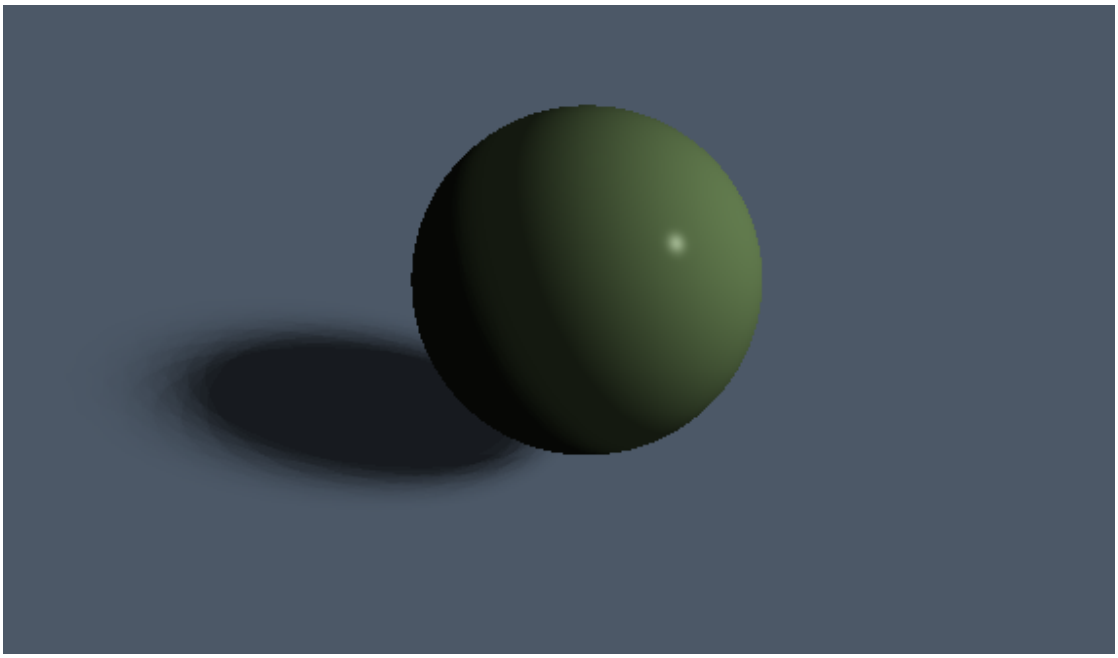

Figure 11: Sphere casting a soft shadow on a surface

References

Agate, M., Grimsdale, R. L., and Lister, P. F. (1991). The hero algorithm for ray-tracing octrees. In *Advances in Computer Graphics Hardware IV*, pages 61–73. Springer.

Barringer, R., Andersson, M., and Akenine-Möller, T. (2017). Ray accelerator: Efficient and flexible ray tracing on a heterogeneous architecture. In *Computer Graphics Forum*, volume 36, pages 166–177. Wiley Online Library.

Bourke, P. (1992). Intersection of a line and a sphere (or circle).

De Greve, B. (2006). Reflections and refractions in ray tracing. *Retrived Oct*, 16:2014.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2012). *Heterogeneous computing with openCL: revised openCL 1*. Newnes.

Glassner, A. S. (1989). *An introduction to ray tracing*. Elsevier.

Howes, L. and Munshi, A. (2015). The opencl specification, version 2.0. *Khronos Group*.

Laine, S., Karras, T., and Aila, T. (2013). Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 137–143.

Li, Y. K. (2012). Parallel physically based path-tracing and shading.

Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28.

Munshi, A., Gaster, B., Mattson, T. G., and Ginsburg, D. (2011). *OpenCL programming guide*. Pearson Education.

Rosen, K. H. (1999). *Handbook of discrete and combinatorial mathematics*. CRC press.

Vasiou, E., Shkurko, K., Mallett, I., Brunvand, E., and Yuksel, C. (2018). A detailed study of ray tracing performance: render time and energy cost. *The Visual Computer*, 34.

Xiao, K., Chen, D., Hu, X., and Zhou, B. (2012). Efficient implementation of the 3d-dda ray traversal algorithm on gpu and its application in radiation dose calculation. *Medical physics*, 39:7619–25.